An Aspect-Oriented Framework for F#

Nitesh Chacowry
Department of Computer Science and Information Systems
Birkbeck, University of London
London, WC1E 7HX, UK

Keith Leonard Mannock
Department of Computer Science and Information Systems
Birkbeck, University of London
London, WC1E 7HX, UK
Email: keith@dcs.bbk.ac.uk
(Contact Author)

Full/Regular Research Paper — (CSCI-ISSE)

***Keywords*—Aspect Oriented Software Engineering, Software Architectures, Agile Software Engineering and Development, Software Testing, Evaluation and Analysis Technologies, Component Based Software Engineering**

***Abstract*—This paper presents the research, design and development of an aspect-oriented framework for F#, a functional programming language developed by Microsoft on the .NET platform[3]. Our framework allows one to insert advice *before*, *after*, or *around* the call to a particular function. We provide two distinct approaches to weaving the advice to the source code: using a *monad-based* weaver, and using a weaver built on *meta-programming* technologies.**

I. Introduction

This paper presents the research, design and development of an aspect-oriented [1][2] framework for Microsoft's implementation of a functional language: F#[3]. Aspect-oriented programming (AOP) is a programming paradigm where functionalities which apply across different modules are cleanly encapsulated into separate components. F# allows the development of programs which are free of side effects, and AOP opposes this notion as it precisely applies side effects to existing programs. There are, however, many use cases for AOP, such as instrumentation, transaction management and caching. By carefully crafting a framework which guarantees that the original computations are unaltered, we obtain a clean separation of concerns, and it remains possible to develop side effect free programs, which can then be extended by transparently weaving in advices.

AOP is usually implemented via a dedicated framework. Such frameworks exist for object-oriented languages. Examples include AspectJ [4] for Java and Policy Injection Application Block [5][6], or Spring.Net [7] for .NET languages.

In Section II we we present a brief overview of aspect-oriented programming (AOP). In Section III, we detail the requirements for the AOP framework we implemented. Section IV details the design choice and presents a technical presentation of our framework. In Section V, we present metrics that contrasts the performance of a regular program with the one in which we have weaved some additional functionality. Finally, Section VI presents some concluding remarks and suggests future work.

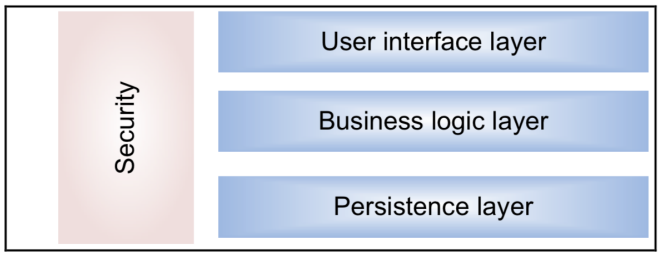

Fig. 1: Horizontal and vertical (cross-cutting) layers in an application

II. Background

The concept of *separation of concerns* (SoC) is an important concept in program design[11]. A common design pattern to achieve separation of concerns in large programs is to encapsulate the concerns into their own distinct layers, where each layer performs a specific functionality [5][12]. For example, an application might be split in three layers: a first layer which handles persistence to a database, a second layer to handle business objects, and a third layer to display data to the user and handle user input. In Figure 1 this functionality is represented by horizontal layers.

However, there maybe additional requirements for including concerns which affect all layers of an application. Such concerns include security, error handling, performance monitoring, thread synchronisation, and transactions[12]. These features are formally referred to as *cross-cutting concerns* [5][13]. In Figure 1 we have modelled the security concern as a vertical layer which affects all other layers. Cross-cutting concerns have the undesirable property that they cause clutter and noise on program code.

AOP is a programming paradigm which allows the encapsulation of these cross-cutting concerns and provides constructs to weave transparently these cross-cutting concerns into working code. A language-neutral definition of AOP is[14]:

> AOP is . . . the desire to make programming statements of the form:
>
> "In program P, whenever condition C arises, perform action A"

AOP consists of the following different components:

Join points

A multitude of events can arise during a programs

execution, such as method calls and exceptions. Join points are the set of events which can arise during the execution of a program[10][13].

Pointcuts
: Pointcuts are used to define the subset of join points on which a specific action should be taken[10][15][16].

Advices
: Advices define the actions that should be taken when a particular join point has been reached during the execution of a program. Referring to the earlier definition, the advice is the action *A* to be performed.

Aspects
: Aspects encapsulate crosscutting concerns [10][15]. Within any AOP framework, aspects store the pointcut and the advice information.

In addition, an advice can be set to execute:

- Before the target function runs. We refer to these as *before advices*.
- After the target function runs. We refer to these as *after advices*.
- Before the target function runs, and also after the function runs. We refer to these as *around advices*.

Referring back to the earlier definition, a program P should have no knowledge of the action A. Clearly, there must be a mechanism to combine the source code of program P with the code defined in advice A. This mechanism is known as *weaving* [1][2][5][10][12] and is illustrated in Figure 2. In the figure there are two aspects, *Aspect A* and *Aspect B*, which are weaved to a source program:

Weaving, the merging of source code with aspects, can be done *statically*, or *dynamically*:

- Static weaving occurs at compile time. In this case the weaver modifies the source code by identifying the selected join points and injecting the advices.
- Conversely, dynamic weaving is a strategy where the weaver inspects running code (or code that is about to be loaded at runtime) and applies advices as specified in the pointcut [10][17].

III. Requirements

The basic requirements we set ourselves for our implementation were:

- The framework was to be developed in F# and advise other programs written in F#.
- Join point: Our join point model would be the execution of functions.
- Pointcut: The framework would provide the ability to select join points by allowing the user to specify the target function. Our framework would allow *named* and *anonymous* pointcuts.
- Advices: the framework would allow the user to specify the additional modules to inject into a target. This requires the implementation of the ability to insert advices: Before, After, and Around.

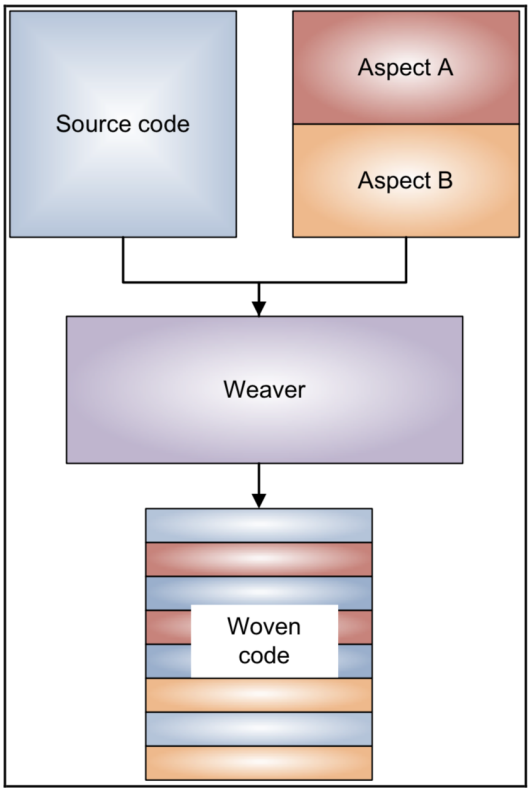

Fig. 2: Weaving aspects to source code

- Our framework has to implement a weaver to inject the advices into the target program. An important design consideration was that the weaver satisfied the concept of *obliviousness*; being (effectively) invisible to the target language.
- Our framework shall allow aspects to be defined such that pointcuts and advices can be grouped.

Figure 3 illustrates the different components of the framework and how they process some user source code (leftmost box "User source code (F#)").

IV. Framework design and implementation

The framework was developed using F# [3][8], and utilised a *test driven approach*[21] using xUnit[22].

A relatively large amount of design and development effort was focused on developing an efficient weaver as it was a central component of the application. Two weaving strategies were investigated and developed:

1) A monad-based weaver, where we shall see that the advice is encapsulated in what is termed a *monadic type*[18]. Monads provide constructs to weave these advices into a program.
2) A weaver based on meta-programming technologies, where advices are implemented as regular F# functions. Special language functionalities are provided to weave the advice function into the source program.

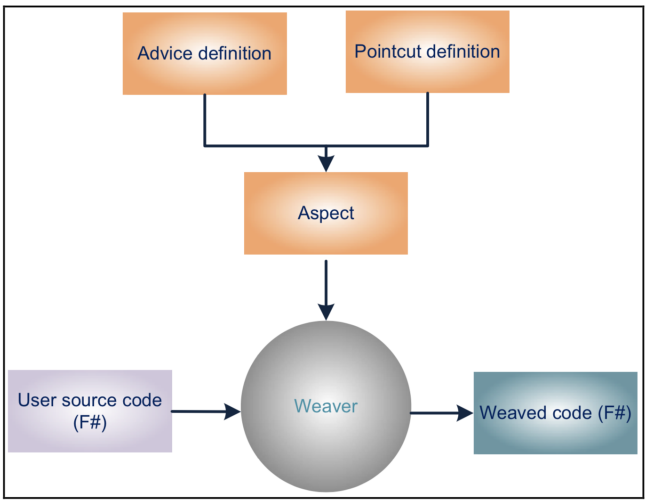

Fig. 3: Framework components

A. Monad-based weavers

The use of monads as a weaving strategy was suggested in [23]. An introduction to monads is given in [24], and [25] provides more information on these programming constructs. Within this section we provide a short introduction to monads, monads in F# and recap the suitability of it as a weaving strategy.

Monads originate from the branch of mathematics known as group theory [26]. In functional languages, a program is usually written such that for a known set of input(s), the program gives one output. This is a fundamental description of functional languages and allows one to reason about a program. However, there are often valid *use cases* which require a program to apply a side effect, for example printing to screen. Monads are often used in functional programming languages to encapsulate these side effects [27], without affecting the original computation or logic of the program.

The core of the monad is the *monadic type*[18]. The purpose of the monadic type is to *augment* or *enhance* the current program, e.g., by applying some relevant side effect. A common notation for the monadic type is $M <' a >$ [18][26] where $'a$ is a generic type representing the *original* type of the computation, and the monadic type $M <' a >$ represents the functionalities/behaviours added by the monad.

Monads further require the use of two operators, known within the literature as the *return* and *bind* operators [18][23]:

- *return* is a first order function which *lifts* the original type $'a$ to the monadic type $M <' a >$, and therefore has the signature $('a \rightarrow M <' a >)$. This function is also known as *unit*[23].
- *bind* is a higher-order function which allows the composition of monadic types together. We define the function signature of bind as:

$$M <' a > \rightarrow ('a \rightarrow M <' b >) \rightarrow M <' b >)$$

Combined together, these operators allow functions to be chained together into a computation. To illustrate, consider the forward pipe operator `|>` (with signature $'a \rightarrow ('a \rightarrow' b) \rightarrow' b$)[28]. We can construct the following computation, if `f(x)` has a signature `(int -> int)` then

```
5 |> f
```

In this example, we are passing in the integer value `5` into the function `f(x)`. We expect the result to be of type `int`, given that `f(x)` accepts an integer and returns another integer. We assume now that there exists a requirement to augment the value `5` (of type integer) to another type $M < int >$. Clearly, we cannot reuse the forward pipe operator, as the function `f(x)` accepts an integer, not a value of the type $M < int >$. However, the solution is to create another function which has the same signature as the bind function, and making the changes to the function `f(x)`:

- Let `f(x)` have the signature `(int -> M<int>)`
- Let *bind* be represented with the notation `>>=`

Then we can reconstruct the original computation chain as follows:

```
return(5) >>= f
```

From [23][27] and the discussion above, monads can be seen as suitable as a weaving strategy as we can use the monadic type $M <' a >$ to encapsulate our advices and use the bind operator to recreate the original computation of the unadvised program. We should also note at this point that within F#, a construct known as *computation expression* (CE)[1], provides syntactic support for monads.

Our implementation uses three single case *discriminated unions*[29] to represent *before*, *after*, and *around* advices. These discriminated unions are concrete implementations of the monadic type $'M < a >$. An F# discriminated union can be compared to the object-oriented construct of an *abstract base class* with a single level of inheritance[18]. A single case discriminated union is therefore analogous to an abstract base class with only one child. The following listing shows the definition of advice types.

```
type BeforeAspect<'T> =
      | Before of 'T

type AfterAspect<'T> =
      | After of 'T

type AroundAspect<'TBefore, 'TAfter> =
      | Around of 'TBefore * 'TAfter
```

Single case discriminated unions are used as they have some advantages when performing pattern matching and result decomposition[18]. The discriminated unions are of a generic type $'T$, which we can use to encapsulate a lambda

[1]Computation expressions are also known as *workflows*[9]. In this paper, we use the term *computation expression* when describing an implementation in F# and the term *monad* to describe the general concept.

function representing the function to execute. Note that the `AroundAspect` is different in that it requires two types: $'TBefore$ and $'TAfter$. In the example below, the advice is a lambda function which takes any input and returns `unit`[20] (`unit` is equivalent to `void` in C# or C++).

```
let beforeaspect =
    Before( fun _ ->
        printfn "running advice.")
```

The `BeforeBuilder` is represented as:

```
type BeforeBuilder() =

member this.Bind(Before(aspectfunc), func) =
  aspectfunc() // execute the advice
  func()
member this.Return(value) = Before(value)
```

The parameters of the `Bind` function are:

- A monadic type which is constrained to the Before case.
- A function with the signature `(unit -> BeforeAspect<'b>)`.

We demonstrate the use of the before computation expression builder by showing how a message is printed to screen before a target function `targetFunction()` executes. The function `targetFunction()` has the signature `(unit -> bool)`:

```
[caption=Sample target function]
let targetFunction() =
      printfn "executing target function."
      true
```

In this case `targetFunction()` does not do anything very interesting, except for printing a message to screen and constantly returning `true`.

A computation expression which uses the computation expression builder is shown below:

```
let before = BeforeBuilder()

let beforeTestRunner(targetfunc) =
    before {
       let! res = beforeaspect
       return targetfunc()
    }

// below is the code to run the CE
beforeTestRunner(targetFunction) |> ignore
```

The computation expression bound to the `beforeTestRunner` identifier (line 3) and accepts a function of signature `(unit -> 'a)`, which matches our concrete implementation `targetFunction()` — which has the more constrained type `(unit -> bool)`.

After running the computation expression, the following output is printed to screen:

```
running advice.
executing target function.
```

A similar approach is used for the `AfterBuilder` computation expression builder.

The around aspect must accommodate the before and after advices being different. The around aspect discriminated union is shown below:

```
type AroundAspect<'TBefore, 'TAfter> =
     | Around of 'TBefore * 'TAfter
```

The single case discriminated union requires a tuple[9], where `'TBefore` is a lambda function representing the before advice, and the `'TAfter` is a lambda function representing the after advice. The computation expression builder is shown in the Listing below:

```
type AroundBuilder() =

 member x.Bind(
    Around(beforeFunc, afterFunc), func) =
    beforeFunc() // call the before aspect

    // call the target function
    // and store the result
    let res = func()

    afterFunc() // call the after aspect
    res // return the result

 member x.Return(value) = Around(value)
```

The `Bind` method performs sequential calls [30] to the before advice (line 5), the target function (line 8) and finally the after advice (line 10). The result of calling the target function is stored in a temporary variable `res` which is returned at the end of the computation. A sample usage is shown below:

```
// Create the advice to execute and
// the aspect

let beforeF = fun _ ->
    printfn "running before advice."

let afterF = fun _ ->
    printfn "running after advice."

let aroundeaspect = Around(beforeF, afterF)

let aroundTestRunner(targetfunc) =
    around {
       let! res = aroundeaspect
       return (targetfunc(), None)
    }

// Running the computation expression.
aroundTestRunner(targetFunction) |> ignore
```

In the Listing shown above, we begin by specifying two

lambda functions which represent the before and after functions. We then create the around aspect which is bound to an identifier called `aroundaspect`. When the computation expression is run, the following expected results are printed to screen:

```
running before advice.
executing target function.
running after advice.
```

From the implementation provided, we note the following advantages when using a monad-based weaver:

- Monads provide clean access to function boundaries.
- Weaving is trivial – the `Bind` method does most of the required weaving without changes to the original computation order.
- Monads allow the encapsulation of any advices — the monadic type $M <' a >$ can encapsulate a wide range of advices.
- Monads are well documented.

We note the following disadvantages when using a monad-based weaving:

- Constructing a pointcut selection language is complex and not intuitive.
- A monad-based approach lacks granularity. To illustrate, consider the following computation which achieves the business requirements of committing some data into a database:

```
let save() = preptxn()
    let transactionResult = executetxn()
    transactionResult
```

  Using the monadic constructs shown previously, it is trivial to attach before, after or around advices to the top level function `save()`. However, we cannot trivially attach an advice to the inner functions called by `save()`. As such, we cannot address a requirement to inject an advice to, say the `executetxn()` function without substantially re-engineering the `save()` function.

We will now (briefly) consider our alternative implementation approach.

B. A Meta-programming based weaver

Meta-programming is a term which refers to a computer program which transforms a source program into another program. This is achieved by treating the source program as input data. Compilers are classical examples of programs that carry out such transformations[19][31]. Conceptually, a weaver performs the same transformation – namely the weaver takes a source, un-advised program, and manipulates it to inject the relevant advices; this behaviour is similar to implementation approaches for Java. In this paper we concentrate on the Monad approach as this proved to be more robust. The meta-programming approach will be described fully in a later publication.

V. Usage and instrumentation

Amongst many examples used to analyse and test the framework, we implemented a version of Machin's forumula for estimating π[37]:

$$\frac{1}{4}\pi = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

We implemented this function in F# and produced two advised versions of the code to judge the performance of our AOP framework: 1) a version using Monads, and 2) a version with the advised code added to the code base manually. For each version we added before, after, and around advices. We then instrumented the code to compare the relative performances and ran the code for k iterations. The results are shown graphically in Figure 4. From the figure we can see that for

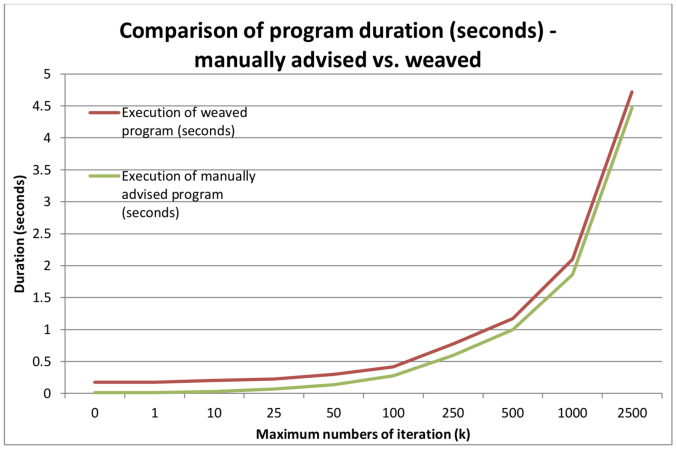

Fig. 4: Manually advised program vs. weaved program

a small number of iterations, it appears that there is almost constant difference between running the manually advised program and the weaved program. This difference also remains constant for larger values of k.

VI. Conclusions and future work

In this paper we have shown how we implemented the components of an AOP framework with the following characteristics:

1) Join points are restricted to function calls.
2) Pointcuts are objects which allow the user to specify which function to advise.
3) Advices are functions to execute either before, after or around the target function.
4) Aspects are an encapsulation of advices and pointcuts.

A key component of our architecture is the weaver. We considered two different designs for our weaver: one using monads and other using meta-programming technologies, namely code quotations. The monad-based weaver is constructed using computation expressions which are unique to F# and provides syntactic support for monad constructs. After experimenting with monads, we decided that this weaver had the better performance and functionality.

We demonstrated the use of our framework to advise a CPU bound program. Some measurements were taken to contrast the time taken to execute a manually advised program with an advised program. This has shown the viability of our approach and the (minor) performance penalty is more than compensated by the *clean code*.

Our future work will include the extension of our pointcut types. We have only provided *kinded pointcuts* which allows one to specify the function name and signature to be advised. AspectJ also provides *non-kinded pointcuts*[10] which inject advices based on the current program flow, lexical context, or execution. The code weaver can be extended to accommodate non-kinded pointcuts as we can reason about the program (e.g., its flow). This may be especially relevant for the abstract syntax tree weaver, as we can readily traverse the tree representing the source program.

REFERENCES

[1] G. Kiczales, J. Lamping, and A. Mendhekar, "Aspect-oriented programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.

[2] G. Kiczales, "Aspect-oriented programming", ACM Computing Surveys, vol. 1241, Dec. 1997, pp. 220-242.

[3] "F# — Microsoft Research", http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.

[4] "The AspectJ Project", `http://www.eclipse.org/aspectj/`.

[5] D. Esposito and A. Saltarello, Microsoft .Net: Architecting Applications for the Enterprise, Microsoft Press, 2015.

[6] "Microsoft Developer Network — Policy Injection Application Block", http://msdn.microsoft.com/en-us/library/ff650672.aspx.

[7] "Spring.NET — Application Framework", http://www.springframework.net/.

[8] "The F# Language Specification", Microsoft Corporation, 2015.

[9] C. Smith, Programming F#, OReilly Media, Inc., Sebastopol CA, 2009.

[10] R. Laddad, AspectJ in action, Manning Publications Co. Greenwich, CT, USA, 2010.

[11] E.W. Dijkstra, "On the Role of Scientific Thought", Selected Writings on Computing: A Personal Perspective, 1982.

[12] K. Baley and D. Belcham, Brownfield Application Development in .Net, Manning Publications Co. Greenwich, CT, USA, 2010.

[13] M. Marin, A.V. Deursen, and L. Moonen, "A Classification of Crosscutting Concerns", Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005

[14] R.E. Filman and D.P. Friedman, "Aspect-oriented programming is quantification and obliviousness", Workshop on Advanced Separation of Concerns, 2000.

[15] D.B. Tucker and S. Krishnamurthi, "Pointcuts and advice in higher-order languages", Proceedings of the 2nd international conference on Aspect-oriented software development — AOSD 03, 2003, pp. 158-167.

[16] "Aspect Oriented Programming with Spring", http://static.springsource.org/spring/docs/2.5.0/reference/aop.html.

[17] "PostSharp — AOP on .Net - Run Time Weaving", http://www.sharpcrafters.com/aop.net/runtime-weaving.

[18] T. Petricek and J. Skeet, Real World Functional Programming, Manning Publications Co. Greenwich, CT, USA, 2010.

[19] K. Hazzard and J. Bock, Metaprogramming in .NET, MEAP Edition Manning, Manning Publications Co. Greenwich, CT, USA, 2012.

[20] R. Pickering, Beginning F#, Apress, 2014.

[21] R. Osherove, The Art of Unit Testing, Manning Publications Co. Greenwich, CT, USA, 2009.

[22] "xUnit — Unit testing framework for C# and .Net (a successor to NUnit)", http://xunit.codeplex.com/.

[23] W.D. Meuter, "Monads as a theoretical foundation for AOP", Technology, 1997, pp. 1-6.

[24] P. Wadler, "The essence of functional programming", Proceedings of the 19th ACM SIGPLANSIGACT symposium on Principles of programming languages, ACM Press, 1992, pp. 1-14.

[25] "Haskell Glossary — Monads", http://www.haskell.org/haskellwiki/.

[26] B. Beckman, "Channel9 : Dont fear the Monad", http://channel9.msdn.com/Shows/Going+Deep/Brian-Beckman-Dont-fear-the-Monads.

[27] S. Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", Engineering theories of software construction, 2001, pp. 47-96.

[28] "Microsoft Developer Network - Symbol and Operator Reference (F#)", http://msdn.microsoft.com/en-us/library/dd233228.aspx.

[29] "Microsoft Developer Network — Discriminated Unions (F#)", http://msdn.microsoft.com/en-us/library/dd233226.aspx.

[30] "Microsoft Developer Network — Expr.Sequential Method (F#)", http://msdn.microsoft.com/en-us/library/ee353459.aspx.

[31] "Meta Programming — Online definition from Cunningham & Cunningham", http://c2.com/cgi/wiki?MetaProgramming.

[32] J. Baker and W. Hsieh, "Runtime aspect weaving through metaprogramming", Proceedings of the 1st international conference on Aspect-oriented software development, ACM, 2002, p. 8695.

[33] . Tanter, R. Toledo, G. Pothier, and J. Noy, "Flexible meta-programming and AOP in Java", Science of Computer Programming, vol. 72, Jun. 2008, pp. 22-30.

[34] G. Kiczales, J. des Rivieres, and D. Bobrow, The Art of the MetaObject Protocol, MIT Press, 1991.

[35] "Microsoft Developer Network — System.Reflection Namespace", http://msdn.microsoft.com/en-us/library/system.reflection.aspx.

[36] J. Liberty, Programming C#, OReilly Media, Inc., Sebastopol CA, 2014.

[37] E.W. Weisstein, "Machins Formula — from Wolfram MathWorld", http://mathworld.wolfram.com/MachinsFormula.html.